The SPARQLGX System for Distributed Evaluation of SPARQL Queries

Damien Graux, Louis Jachiet, Pierre Genevès, Nabil Layaïda

The SPARQLGX System for Distributed Evaluation of SPARQL Queries

Damien Graux, Louis Jachiet, Pierre Genevès, and Nabil Layaïda

Abstract—SPARQL is the W3C standard query language for querying data expressed in the Resource Description Framework (RDF). The increasing amounts of data available in the RDF format raise a major need and research interest in building efficient and scalable distributed SPARQL query evaluators. In this context, we propose SPARQLGX: an implementation of a distributed RDF datastore based on Apache Spark. SPARQLGX is designed to leverage existing Hadoop infrastructures for evaluating SPARQL queries efficiently. SPARQLGX relies on an automated translation of SPARQL queries into optimized executable Spark code. We show that SPARQLGX makes it possible to evaluate SPARQL queries on billions of triples distributed across multiple nodes, while providing attractive performance figures. We report on experiments which show how SPARQLGX compares to state-of-the-art implementations and we show that our approach scales better than other systems in terms of supported dataset size. With its simple design, SPARQLGX represents an interesting alternative in several scenarios.

Improvements— A preliminary version of this work was presented at the ISWC 2016 conference [1]. Since then, the SPARQLGX system has been improved in many aspects. Compared to [1], the present article provides two notable additions: (i) a detailed presentation of the system architecture with its most important concepts *i.e.* the various adopted storage strategies and their benefits, the supported SPARQL fragment (which has been extended with general disjunctions, safe optionals and safe conjunctions), the query translation process including examples, and (ii) an extensive experimental analysis where SPARQLGX is systematically compared with a panel of nine third-party state-of-the-art systems.

Index Terms—RDF, SPARQL evaluation, distributed systems, comparative evaluation.

1 Introduction

SPARQL is the standard query language for retrieving and manipulating data represented in the Resource Description Framework (RDF) [2]. SPARQL constitutes one key technology of the semantic web and has become very popular since it became a W3C recommendation [3].

The construction of efficient SPARQL query evaluators faces several challenges. First, RDF datasets are increasingly large, with some already containing more than a billion triples. For querying efficiently such amounts of data, evaluators need to be distributed and to scale. Furthermore, semantic data might exhibit a dynamic nature, as they are subject to change. Thus being able to answer quickly after a change in the input data also constitutes an interesting property for a SPARQL evaluator. In this context, we propose SPARQLGX: an engine designed to evaluate SPARQL queries efficiently on large distributed RDF datasets. SPARQLGX relies on a compiler of SPARQL conjunctive queries into optimized Scala code that is executed by the Apache Spark [4] backend. SPARQLGX is open-source and available from: `https://github.com/tyrex-team/sparqlgx`

This article is organized as follows: we first introduce some required preliminary knowledge in Section 2. Then, in Section 3, we describe the most important concepts used in the design and implementation of SPARQLGX. Section 4 reports on our experimental evaluation where we compare our implementation with other open source HDFS-based RDF systems. Finally, we review related works in Section 5 and conclude in Section 6.

- *Authors' affiliations: Inria, LIG, CNRS, Université Grenoble Alpes.*

- *This study has been partially supported by the Datalyse and CLEAR projects.*

Manuscript sent: October 23, 2017.

2 Background

The Resource Description Framework (RDF) is a language standardized by W3C to express structured information on the Web as graphs [2]. It models knowledge about arbitrary resources using Unique Resource Identifiers (URIs), Blank Nodes and Literals. RDF data is structured in sentences – or triples– written (s p o), each one having a subject s, a predicate p and an object o.

Hadoop is a framework for distributed system based on the Map-Reduce paradigm [5], it is used by numerous evaluators. Hadoop consists of two main parts: the Hadoop distributed file system (HDFS) [6] and a MapReduce library for distributed processing. Hadoop consists of a number of different daemons/servers: NameNode, DataNode, and Secondary NameNode for managing HDFS, and JobTracker and TaskTracker for performing MapReduce. Technically, Files in HDFS are split into a number of large blocks (usually a multiple of 64MB) which are stored on DataNodes. A file is typically distributed over a number of DataNodes in order to facilitate high bandwidth and parallel processing. In order to improve reliability, data blocks in HDFS are replicated and stored on three (default parameter) machines, with one of the replicas in a different rack for increasing availability further. The maintenance of file metadata is handled by a separate NameNode. Such metadata includes mapping

from file to block and location (DataNode) of block. The NameNode periodically communicates its metadata to a Secondary NameNode which can be configured to do the task of the NameNode in case of the latters failure.

Apache Spark [4] is a MapReduce-like data-parallel framework designed for large-scale data processing running on top of the JVM. Spark can be set up to use HDFS.

3 SPARQLGX Main concepts

In this Section, we present the main design concepts and implementation techniques used in SPARQLGX. We first review design choices made in representing and storing the data model. We then present important optimizations for improving the performance of the data loading phase. Finally we explain how we translate queries from the supported SPARQL query language fragment into optimized lower-level Scala code [7] which is directly executed with the Spark API.

3.1 Data Storage Model: Vertical Partitioning

From a "raw" storage (*e.g.* a file in the N-Triples standard [8] which is a simple list of all triples) to complex schemes (*e.g.* involving indexes or B-trees), there are many ways to store RDF data. Any storage choice is a compromise between **(1)** the time required for converting origin data into the target format, **(2)** the total disk-space needed, **(3)** the possible response time improvement induced.

RDF triples have very specific semantics. In a RDF triple ($s\ p\ o$), the predicate p represents the "semantic relationship" between the subject s and the object o. Thus, there are often relatively few distinct predicates compared to the number of distinct subjects or objects. The vertically partitioned architecture introduced by Abadi *et al.* in [9] takes advantage of this observation by storing the triple ($s\ p\ o$) in a file named p whose contents keeps only s and o entries. This approach has the following advantages and characteristics:

(1) Converting RDF data into a vertically partitioned dataset does not involve complex computation: each triple is read once and the pair (subject, object) is appended to the predicate file.

(2) For large datasets with only a few predicates, two values (a value is a URI or a blank node or a literal) are stored instead of three which reduce the memory footprint compared with the input dataset.

(3) Having vertically partitioned data reduces evaluation time of triple patterns whose predicate is a constant (*i.e.* not a variable): searches are limited to the relevant files. In practice, one can observe that most SPARQL queries have triple patterns with a constant predicate. [10] showed that graph patterns where all predicates are constant represent 77.81% of the queries asked to DBpedia [11] and 98.08% of the ones asked to SWDF.

We consider that vertical partitionning is very well suited in the context of distributed evaluation over RDF data without reasoning: it implies a pass over the data but with only simple computations, it reduces the size of the dataset and provides an indexation. This is the storage model that we adopt on HDFS, and with which we make RDF datasets available to Apache Spark.

3.2 Loading Optimizations

3.2.1 Compression

In the vertical partitioning scheme, we have to store for each triple pattern (TP) a subject and an object. In real datasets, URIs are strings of 20 to 100 characters and are heavily redundant. It seems therefore rather natural to compress these URIs. RDF datasets are prone to efficient compression [12]. Yet, while applying an efficient text compression algorithm on the whole dataset would be very effective in terms of compression ratio, it would be less effective for the query evaluation process.

If we compress the subjects and objects, we have several benefits: data take up less disk space and reading the dataset takes less time during the evaluation. In addition, if we can also leave these URIs compressed during the evaluation, the data exchanged for the join computations would also be compressed which means faster shuffles and in the case of large output, this also means faster output.

Most standard compression schemes usually operate by building some kind of frequency table while reading the input and thus have very bad compression ratio if we simply apply them on small strings (such as URIs). In SPARQLGX we rely on a double-step compression: prefix compression and gzip compression.

3.2.2 Prefix compression

The first step is a simple yet efficient and very fast scheme. Given a dataset, we compute a set of prefixes $P = p_1, \ldots, p_n$ and given a URI $< u >$ we find the longest p_i such that $u = p_i u'$ and replace $< u >$ with $i : u'$. When such a prefix does not exist, $< u >$ is left unchanged.

We compute P using two parameters (see Algorithm 1): U controls the maximum size allowed for a prefix, and k controls the total number of prefixes. P is computed iteratively in $log_2(U)$ map-reduce steps. At each step we compute a new set P' of prefixes such that each element of P' is the longest prefix (among P') of at least k URIs. The size of P' is thus bounded by the number of URIs L divided by k. P is the union of the P' therefore P contains in the worst case $L \times log_2(U)/k$ elements. However, since at each step we only keep prefixes such that there are L/k elements with this prefix, we can expect on real data to have much less prefixes (around k at most).

3.2.3 Gzip compression

The second step is done when the data is about to be stored on disk, in plain files containing pairs of subject-object. When Hadoop stores larges files, it splits them into chunks, we compress each of those chunks using gzip. While gzip is not the best compression algorithm (in terms of compression ratio) it is standard and has a very fast decompression speed; which means reading the input will not be CPU bounded.

3.2.4 HDFS Optimization

Data are stored by HDFS as gzipped files. With HDFS, we pay a little overhead for each chunk of each file. Furthermore, for gzip to compress efficiently, we also need the chunks to have a reasonable size. For these reasons we control the number of chunks so that each predicate gets

Algorithm 1 Computation of prefixes

$L \leftarrow$ list of URIs
$U \leftarrow$ maximum size of a prefix
$K \leftarrow$ number of URIs per prefix

function STEP(P, L, u)
  $count \leftarrow$ empty map
  for $l \in L$ **do**
    $prefix \leftarrow$ longest $p \in P$ prefix of l
    if $|l| \geq |prefix| + u$ **then**
      $newPrefix \leftarrow substring(l, |prefix| + u)$
      $count[newPrefix] += 1$
  end for
  $res \leftarrow \{v \in key(count) \mid count[v] \geq K\}$
  return res
end function

function COMPUTEPREFIX(L, U, K)
  $p \leftarrow \{""\}$
  $u \leftarrow U/2$
  while $u \neq 0$ **do**
    $p \leftarrow p \cup step(p, u)$
    $u /= 2$
  end while
  return p
end function

a number of chunks proportional to the number of triples it corresponds to.

3.3 Respective Optimizations in Practice

It is worthwhile to comment on the combined effect of the aforementioned design choices and optimizations in practice. We thus review the respective gains brought by their successive application on three popular datasets: WatDiv [13], LUBM [14] and DBpedia [15].

In our tests, we choose $U = 128$ and $k = 5000$. Choosing $U = 128$ seems a good fit: we set U to be a power of 2 (since the algorithm always runs in $\lceil log_2(U) \rceil$ map reduce phases) and most URIs contains less then 128 characters (thus $U \leq 128$) but many contains more than 64 characters. Choosing $k = 5000$ is more arbitrary: as k increases, we add to the set of prefixes new real prefixes (such as domain names), but if k is big enough our method arrives at a point where the new prefixes it discovers share long common prefixes and increasing k leads to very small benefits in terms of compression.

For instance, DBpedia [15] stores many of its object using URIs of the form `http://wikidata.dbpedia.org/resource/Q<id>`. Ideally our method would just add the `.../Q` part as the `<id>` are already in a thigh representation (decimal numbers) but in practice our method will add `...Q1` to `...Q9` and if k increases it will add `Q10` to `Q99`, etc. Notice that, as k increases, the set will continue to be more and more compressed but the marginal gain will tend to zero while the computational cost will increase. With $k = 5000$ our method find 2135 prefixes starting with `http://wikidata.dbpedia.org/resource/Q`. That is why we set $k = 5000$, high enough so that datasets are well compressed, but low enough for the prefix file to be very small (133 KB for DBpedia with $k = 5000$ which is less than a millisecond of transfer time in a gigabit world).

The absolute theoretical maximal number of prefixes is $k \times log(U) = 35000$ but, as expected, we have much less prefixes in practice: 2319 for Lubm10k, 1764 for WatDiv10k and 3119 for DBpedia.

Table 1 reports on the overall effect of each compression mechanism. We see that on WatDiv the prefixing part of our compression scheme compresses the data a lot despite a relatively low number of prefixes: we attribute this compression to the shape of the generated data. On Lubm, it is quite the opposite, the generated URIs start in many different ways; for the gzip-ped version, it is the most compressed dataset, which is because the data is redundant. DBpedia is the least compressed, it is because there are 2149 different predicates in DBpedia, many of them corresponding to a few triples patterns and the gzip-ping of small files is not very efficient.

Finally, even for a real-world dataset like DBpedia, we end up storing around one tenth of the original dataset on disk and our prefixization reduces the size of the data that we manipulate during query evaluation by approximately 63% on average.

Datasets	Initial	VP		VP+Prefixed		VP+Prefixed+Gz	
WatDiv10k	149 GB	102 GB	68%	21 GB	14%	11GB	7%
Lubm10k	232 GB	150 GB	65%	71 GB	31%	12 GB	5%
DBpedia	181 GB	125 GB	69%	45 GB	25%	19 GB	10%

TABLE 1: Respective dataset sizes after the successive application of SPARQLGX's data representation and loading techniques.

3.4 Supported SPARQL Fragment

SPARQLGX supports the SELECT fragment of the SPARQL query language with modifiers (see section 3.5.6) and where the graph pattern is a query composed of triple patterns, conjunctions, disjunctions and optionals but where conjunctions (resp. optionals) are restricted to *safe* conjunctions (resp. *safe* optionals). A formal semantics of the SPARQL language is given in [16]. The syntax of the supported graph patterns is given in Figure 1 and we now define the *safe* criterion.

A query contains variables and a *query solution* is a mapping from some of the query variables towards values. Some of query variables *must* appear in all the solutions while others *might* be present or not. For each variable we can determine whether it *must* appear in all solutions or whether it *might* appear only in some of the solutions.

A conjunction (resp. an optional) between two subqueries A and B is *safe* when all the query variables that might appear in a solution of A and in a solution of B must appear in all solutions of A and B.

See figure 2 for a formal definition of variable that *must* or *might* appear and of safe JOINs and OPTIONALs.

Notice when a term contains no disjunction nor optional all its variables are sure to appear. Our fragment thus contains the Basic Graph Pattern (BGP) fragment of SPARQL (conjunction of TP), extensively studied in the literature. The

Query :=
- TP
- Query JOIN Query
- Query OPTIONAL Query
- Query UNION Query

TP := UV UV UV

UV :=
- ?variable
- <uri>
- "litteral"

Fig. 1: Syntax of supported SPARQL queries.

$$
\begin{aligned}
must(Q_1 \text{ JOIN } Q_2) &:= must(Q_1) \cup must(Q_2) \\
must(Q_1 \text{ OPTIONAL } Q_2) &:= must(Q_1) \\
must(Q_1 \text{ UNION } Q_2) &:= must(Q_1) \cap must(Q_2) \\
must(TP) &:= vars(TP) \\
\\
might(Q_1 \text{ JOIN } Q_2) &:= might(Q_1) \cup might(Q_2) \\
might(Q_1 \text{ OPTIONAL } Q_2) &:= might(Q_1) \cup might(Q_2) \\
might(Q_1 \text{ UNION } Q_2) &:= might(Q_1) \cup might(Q_2) \\
might(TP) &:= vars(TP)
\end{aligned}
$$

$$
safe(Q_1 \text{ OPTIONAL } Q_2) = safe(Q_1 \text{ JOIN } Q_2) = \\
(might(Q_1) \cap might(Q_2) = must(Q_1) \cap must(Q_2))
$$

Fig. 2: Safe terms.

optional of BGPs is also comprised in our fragment. Given two queries in our fragment, their union is always in our fragment as union does not need to be safe.

3.5 SPARQL Fragment Translation

We now review the main principles with which SPARQL queries are compiled into executable Spark code. A query is translated recursively into Spark code which is in charge of computing query solutions. We now present how to translate the various elements of our language.

3.5.1 Storage of partial solutions

The partial solutions to a query Q are stored as a list (an RDD in Spark terminology) of n-uplets where n is the number of variables that might appear in Q. These RDDs are eventually partitioned along a key and given two RDDs partitioned by keys, we can efficiently compute the join (or the left-join) on their keys.

From Spark point of view, RDDs are just n-uplets (with eventually a key that is another k-uplet). During the compilation we thus maintain an association between the i-th component of an n-uplet and the variable it corresponds to.

Since partial solutions are all n-uplets but might not have all the variables that might appear, we use the empty string (*i.e.* `""`) for the value of a variable that is not bound.

3.5.2 Triple Patterns

To compute the solutions for a unique TP: when the predicate is a constant, the relevant HDFS file is simply opened with a call to Spark's `textFile` method; otherwise, we have to open all predicate files. Then, using the constants of the TP, we use a `filter` to keep only the pairs (subject,object) corresponding to matching triples and `map` to keep only the parts of TPs corresponding to variables. For instance, the TP {?s age 21 .} matching people that are 21 years old is translated into:

```
val tp=sc.textFile("age.txt")
         .filter{case(s,obj)=>obj==21}
         .map{case (s,obj) => s}
```

3.5.3 Conjunctions

In order to translate a conjunction of two subqueries Q_1 and Q_2, each Q_i is first translated. Since conjunctions are all *safe* (see Section 3.4), the set of common variables between two mappings of Q_1 and Q_2 is known at compile time. Therefore the conjunction can be performed using a classic join using their common variables as a key.

When needed, we can use `keyBy` in Spark to repartition both partial solutions along their common variables. The materialization of the conjunction is realized with the `join` keyword. After the join the resulting RDD is a triple of the key used, the columns from Q_1 and the columns from Q_2. We reshape the RDD using the keyword `mapValues` to inform spark that the partitioning is not invalidated since we only changed the values.

For example the TPs {?s age 21 . ?s gender ?g .} are translated into:

```
val tp1=sc.textFile("age.txt")
      .filter{case(s,obj)=>obj==21}
      .map{case (s,obj) => s}
      .keyBy{case s=>s}
val tp2=sc.textFile("gender.txt")
      .keyBy{case(s,g)=>s}
val bgp=tp2.join(tp1)
      .mapValues{
        case((tp2s,tp2g),(tp1s))=>(tp2s,tp2g)
       }
```

A join with no common variables corresponds to a cross product (`cartesian` in Spark).

3.5.4 Optionals

A query of the form Q_1 OPTIONAL Q_2 can be translated in the following way. Since the optionals are *safe* (as for conjunctions) there is a set of common variables. Once both Q_i are translated, we perform a left join between the translation of Q_1 and Q_2.

Just like for conjunction we first apply an eventual `keyBy` on the translation of both Q_i then we do a `leftOuterJoin` and then we need to reshape with a `mapValues`. However contrary to the joins in the `mapValues` we need to account for columns that are present multiple times but also for the fact that the Q_2 might be missing.

The query {?s age 21 OPTIONAL ?s gender ?g .}, for example, is translated into:

```
val opt=tp1.leftOuterJoin(tp2)
 .mapValues{
   case((tp1s),None)=>(tp1s,"")
   case(tp1s,Some((tp2s,tp2g)))=>(tp1s,tp2g)
  }
```

3.5.5 Disjunctions

For disjunctions, we need to first translate both subqueries, discard the keys if they don't match, extend (with the value "") and re-order the set of columns of each subquery so they have the same set of columns in the same order and finally return the union using the keyword `union`.

The query {?s age 21 UNION ?s gender ?g .}, for example, is translated into:

```
val extended_tp1 = tp1.map{case s => (s,"")}
val uni = extended_tp1.union(tp2)
```

3.5.6 Modifiers

Once the query is translated, we use a `map` to retain only the desired fields (*i.e.* the distinguished variables) of the query. At that stage, we can also modify results according to the SPARQL solution modifiers [3] (*e.g.* removing duplicates with `distinct`, sorting with `sortByKey`, returning only few lines with `take`, *etc.*)

The obtained translation (the Scala/Spark code) thus depends on the initial order of TPs since the joins will be performed in the same order. This is prone to optimizations based on join commutativity which we now present.

3.6 Optimized Join Order With Statistics

The evaluation process (using Spark) first evaluates TPs and then joins these subsets according to their common variables; thus, minimizing the intermediate set sizes involved in the join process reduces evaluation time (since communication time between workers is reduced). Thereby, statistics on data and information on intermediate results sizes provide useful information that we exploit for optimisation purposes.

Given an RDF dataset $\mathcal{D}$ having T triples, and given a place in an RDF sentence $k \in \{subj, pred, obj\}$, we define the selectivity in $\mathcal{D}$ of an element e located at k as: (1) the occurrence number of e as k in $\mathcal{D}$ if e is a constant; (2) T if e is a variable. We note it $sel^k_{\mathcal{D}}(e)$. Similarly, we define the selectivity of a TP (a b c .) over an RDF dataset $\mathcal{D}$ as: $SEL_{\mathcal{D}}(a,b,c) = \min(sel^{subj}_{\mathcal{D}}(a)\,,\, sel^{pred}_{\mathcal{D}}(b)\,,\, sel^{obj}_{\mathcal{D}}(c))$.

Thereby, to rank each TP, we compute statistics on datasets counting all the distinct subjects, predicates and objects. This is implemented in a compile-time module that sorts TPs in ascending order of their selectivities before they are translated.

Finally, we also want to avoid cartesian products. Given an ordered list l of TPs we compute a new list l' by repeating the following procedure: remove from l and append to l' the first TP that shares a variable with a TP of l'. If no such TP exists, we take the first.

Before translating a query, we use an additional module that implements this idea in order to sort the TPs in ascending order of their selectivities.

3.7 SDE: SPARQLGX as a Direct Evaluator

Our tool evaluates SPARQL queries using Apache Spark after preprocessing RDF data. However, in certain situations, data might be dynamic (*e.g.* subject to updates) and/or users might only need to evaluate a single query (*e.g.* when evaluation is integrated into a pipeline of transformations). In such cases, it is interesting to limit as much as possible both the preprocessing time and the query evaluation time.

To take the original triple file as source, we only have to modify in our translation process the way we treat TPs to change our storage model. Instead of searching in predicate files, we directly use the initial file; and the rest of the translation process remains the same. We call this variant of our evaluator the "direct evaluator" or SDE.

4 Extended Evaluation

In this Section, we report on extensive practical experiments made with SPARQLGX, and provide a detailed comparative analysis with other state-of-the-art evaluators. This provides a fresh perspective on distributed SPARQL evaluators, based on a multi-criteria ranking obtained through extensive experiments. Specifically, we propose a set of five principal features (namely velocity, immediacy, dynamicity and resiliency) which we use to rank evaluators. Each system exhibits a particular combination of rankings accross these features.

Our suggested set of features provides a more comprehensive description of the behavior of a distributed evaluator when compared to traditional performance metrics. We show how it helps in more accurately evaluating to which extent a given system is appropriate for a given use case. For this purpose, we systematically benchmarked a panel of 10 state-of-the-art implementations. We ranked them using this reading grid to pinpoint the advantages and limitations of SPARQLGX and the current competing SPARQL evaluation systems.

4.1 Methodology For Experiments

For studying how well the distribution techniques perform, we tested the 10 systems presented in Section 4.2 with queries from two popular benchmarks (LUBM and WatDiv), which we evaluated on several datasets of varying size. We precisely monitored the behavior of each system using several metrics encompassing *e.g.* total time spent, CPU and RAM usage, as well as network traffic. In this Section, we describe our experimental methodology in further details.

4.1.1 Datasets and Queries

As introduced in Section 3.4, we focus here on the Basic Graph Pattern (BGP) fragment which is composed of the set of conjunctive queries. It is also the common fragment supported by all tested stores and thus provides a fair and common basis of comparison.

Also for a fair comparison of the systems introduced in Section 4.2, we decided to rely on third-party benchmarks. The literature about benchmarks is also abundant (see *e.g.* [17] for a recent survey). For the purpose of this study, we selected benchmarks according to two conditions: (1) queries should focus on testing the BGP fragment and (2)

the benchmark must be popular enough in order to allow for further comparisons with other related studies and empirical evaluations (such as [18] for instance). In this spirit, we retained the LUBM benchmark[1] [14] and the WatDiv benchmark[2] [13].

LUBM is composed of two tools: a determinist parametric RDF triples generator and a set of fourteen queries. Similarly, WatDiv offers a determinist data generator which creates richer datasets than the LUBM one in the sens of the number of classes and predicates, in addition, it also comes with a query generator and a set of twenty query templates. We used several standard LUBM and WatDiv datasets with varying sizes to test the scalability of the compared RDF datastores. Table 2 presents the characteristics of datasets we used. We selected in particular these three ones because they are gradually RAM-limiting: the WatDiv1k dataset can be held in memory of one single VM, the Lubm1k dataset becomes too large and Lubm10k is larger than the whole available RAM of our cluster.

Datasets	Number of Triples	Size
WatDiv1k	109 million	15 GB
Lubm1k	134 million	23 GB
Lubm10k	1.38 billion	232 GB

TABLE 2: Size of sample datasets.

We evaluated on these datasets the provided LUBM queries and generated the WatDiv queries according to the provided templates. LUBM queries (Q1-Q14) were made to represent real-world queries while remaining in the BGP fragment of SPARQL and with a small data complexity (the size of the answer for a query is always almost linear in the size of the dataset). In addition, in the LUBM query set, we notice that one query is challenging: Q2 since it involves large intermediate results and implies a complex join pattern called "triangular". WatDiv queries compared with LUBM ones involved more predicates and classes. Furthermore, WatDiv developers already group query templates according to four categories: linear queries (L1-L5), star queries (S1-S7), snowflake-shaped queries (F1-F5) and complex queries (C1-C3).

In addition, we can represent a BGP query by a graph where each node corresponds to a triple pattern and where edges between nodes represent a common variable. As presented respectively in Tables 3 & 4, LUBM and WatDiv queries can be grouped according to their variable graphs. Moreover, the WatDiv query graphs (Table 4) show alternate grouping methods – *i.e.* C3, F2 and F4 are all variations around an hexagonal graph – than the one presented in [13].

4.1.2 Metrics

During our tests we monitored each task by measuring not only time spent but a broader set of indicators:

1) *Time (Seconds)*: simply measures the time taken by the system to complete a task.
2) *Disk footprint (Bytes)*: measures the use of disks for a given dataset size including indices and any auxiliary data structures.

1. http://swat.cse.lehigh.edu/projects/lubm/
2. http://dsg.uwaterloo.ca/watdiv/

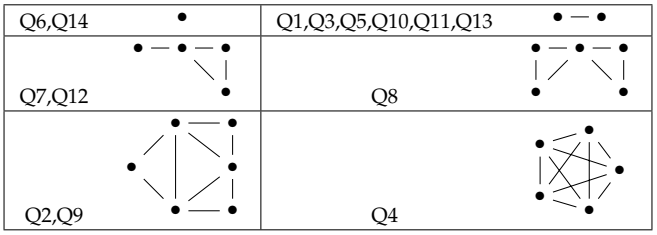


TABLE 3: Variable graphs associated to LUBM queries.

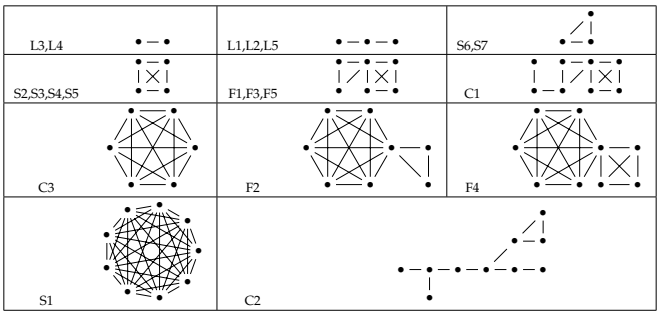


TABLE 4: Variable graphs associated to WatDiv queries.

3) *Disk activity (Bytes/second)*: measures at each instant the amount of bytes written on and read from the disks during processes.
4) *Network traffic (Bytes/second)*: measures how much data is exchanged between nodes in the cluster.
5) *CPU usage (percentage)*: measures how much the CPU is active during the computation.
6) *RAM usage (Bytes)*: measures how much the RAM is used by the computation.
7) *SWAP usage (Bytes)*: measures how much SWAP is used. Such a metric will be particularly measured when the system runs out of RAM and thus be often omitted.

4.1.3 Cluster Setup

Our experiments were conducted on a cluster composed of Virtual Machines (VMs) hosted on two servers. The first server has two processors Intel(R) Xeon(R) CPU E5-2620 cadenced at 2.10 GHz, 96 GigaBytes (GB) of RAM and hosts five VMs. The second server has two processors Intel(R) Xeon(R) CPU E5-2650 cadenced at 2.60GHz with 130 GB of RAM and hosts 6 VMs: 5 dedicated to the computation (like the 5 VM of the first server) plus one special VM that orchestrates the computation. Each VM has dedicated 2 physical cores (thus 4 logical cores), 17 GB of RAM and 6 TeraBytes (TB) of disk. The network allows two VMs to communicate at 125 MegaBytes per Seconds (MB/s) but the total link between the two servers is limited at 110 MB/s. The read and write speeds are 150 MB/s and 40 MB/s shared between the VM on the first server and 115 MB/s and 12 MB/s shared between the VM of the second server.

4.1.4 Extensive Experimental Results

We made our extensive experimental results openly available online[3] with more detailed information. In particular, for reproducibily purposes, we wrote tutorials on how to

3. http://tyrex.inria.fr/sparql-comparative/home.html

install and configure the various tested evaluators and report all the versions of the systems we used. We also share measurements and graphs for all the considered metrics and for each node.

In the rest of the paper, we focus on summarizing and discussing the essence of the lessons that we learned from our experiments. In Section 4.3 we report on the overall behavior of each system pushed to the limits during the tests. In Section 4.4 we further discuss and develop a comparative analysis guided by practical features that imply different requirements.

4.2 Benchmarked datastores

In addition to the evaluators we proposed (*i.e.* SPARQLGX and SDE in their first version), we also benchmarked several popular evaluators that we briefly describe here focusing on their particularities for supporting RDF querying.

We used several criteria in the selection of the SPARQL evaluators tested. First, we choose to focus on distributed evaluators so that we can consider datasets of more than 1 billion triples which is larger than the typical memory of a single node in a commodity cluster. Furthermore, we retained systems that support at least a minimal fragment of SPARQL composed of conjunctive queries and called the BGP fragment (further detailed in Section 3.4). We focused on open-source systems. We wanted to include some widely used systems to have a well-known basis of comparison, as well as more recent research implementations. We also wanted our candidates to represent the variery and the richness of underlying frameworks, storage layouts, and techniques found – see *e.g.* taxonomies of [19] and [20] –, so that we can compare them on a common ground. We finally selected a panel of 10 candidate implementations, presented in Table 5.

Table 5 also summarizes the characteristics of the systems we used in our tests. We split our panel of 10 implementations into subcategories. The first category, called *standalone systems*, gathers systems that distribute data using their own custom methods. In contrast, all the other systems use the well-known HDFS distributed file system [6] for this purpose. HDFS handles the distribution of data across the cluster and its replication. It is a tool included in the Apache Hadoop[4] project which is a framework for distributed systems based on the MapReduce paradigm [5].

We further subdivide the HDFS-*based systems* into two categories: the *preprocessing-based evaluators* and the *direct* SPARQL *evaluators*. The first category requires some preprocessing whereas direct SPARQL evaluators use distributed data without preprocessing. We first summarize some required background on SPARQL and then further review the candidates of each category below.

4.2.1 Standalone Datastores

4store: 4store[5] is a native RDF solution introduced in [21]. 4store has an index to translate URIs to identifiers, which allows a space-efficient representation of triples. For each predicate it uses two indexes (subject to object and object to subject) for optimizing query evaluation. 4store distinguishes two types of cluster nodes: some nodes only store data while others are responsible for parsing, communicating with storage nodes and aggregating the results.

CumulusRDF: CumulusRDF[6] [22] relies on Apache Cassandra[7] [23] and mixes two strategies: indexing and hashing. Each triple is hashed and distributed through Cassandra. Additionally the CumulusRDF layer computes indexes to optimize the search of triples satisfying TPs.

CouchBaseRDF: CouchBase[8] is not a native RDF solution but a document-oriented NoSQL database system, well-known in the NoSQL world. The specificity of this datastore is that it adopts an in-memory approach where a dataset is distributed on the main memory of the cluster's nodes. This is a limitation because the whole dataset has to fit inside the global RAM – but this speeds up query evaluation. Querying is done by MapReduce rounds on CouchBase controlled by Apache Jena[9], which optimizes the execution plan. CouchBaseRDF [18] transforms CouchBase into an RDF solution. It maps the RDF triples onto JSON documents, each document corresponds to a subject and contains two JSON arrays of the same size: the predicates and the objects. This encoding is used to optimize the retrieval of triples when the subject is fixed. Three views are pre-generated to cover other TPs (when predicate, object or both are fixed values).

4.2.2 HDFS-based Datastores

RYA: RYA [24] is a native RDF solution leveraging Apache Accumulo that creates three indexes and stores them in Accumulo. Accumulo then sorts and partitions these tables across the nodes, storing data on the HDFS.

S2RDF: S2RDF [25] uses SparkSQL to store RDF triples. SparkSQL [26] is a library built to leverage relational data on top of Apache Spark [4]. It allows users to register files as tables and then to query them using the SQL relational query language. It thus offers a way to set up a distributed relational store, potentially leveraging years of research in relational database systems. S2RDF adopts the vertical partitioning [9] to construct its tables and also computes additional tables based on pre-computation of possible joins representing co-occurrence of a variable in two different fields. Before the evaluation, S2RDF translates SPARQL queries into SQL ones using statistics on orignal data (generated during the preprocessing phase) to order joins by selectivity.

CliqueSquare: CliqueSquare [27] is a native RDF solution. The specificity of CliqueSquare lays in trying to reduce the response time by flattening execution plans. Specifically, it implements optimizations whose goal is to minimize the height of the tree of joins in execution plans. It does so in the query optimization phase but also in the way it stores data. Each node is responsible for a set of values storing all triples containing these values as subject, predicate or object (a triple is thus stored, at most, thrice).

PigSPARQL: PigSPARQL [28] compiles a SPARQL fragment to PigLatin [29], which is a programming language

4. http://hadoop.apache.org/
5. http://4store.org/
6. http://code.google.com/p/cumulusrdf/
7. http://cassandra.apache.org/
8. http://www.couchbase.com/
9. https://jena.apache.org/

	Systems	Underlying Framework	Storage Back-End	Storage Layout	SPARQL Fragment
Standalone Datastores	4store	—	Data Fragments	Indexes	SPARQL 1.0
	CumulusRDF	Cassandra	Key-Value store	3 hash and sorted indexes	SPARQL 1.1
	CouchBaseRDF	CouchBase	Buckets	3 views	Basic Graph Pattern
HDFS-based Datastores with preprocessing	RYA	Accumulo	Key-Value store on HDFS	3 sorted indexes	Basic Graph Pattern
	SPARQLGX	Spark	Files on HDFS	Vertically Partitioned Files	Basic Graph Pattern
	S2RDF	SparkSQL	Tables on HDFS	Extended Vertically Partitioned Files	Basic Graph Pattern
	CliqueSquare	Hadoop	Files on HDFS	Indexes	Basic Graph Pattern
HDFS-based Direct Evaluators	PigSPARQL	PigLatin	Files on HDFS	N-Triples Files	SPARQL 1.0
	RDFHive	Hive	Relational store on HDFS	Three-column Table	Basic Graph Pattern
	SDE	Spark	Files on HDFS	N-Triples Files	Basic Graph Pattern

TABLE 5: Systems used in our tests.

for distributed systems. PigSPARQL has no actual loading phase. It reads its data directly from the HDFS in the N-Triples W3C standard [8] (*i.e.* a plain text file, with one triple per line with space as the field separator). The PigSPARQL compilation tries to optimize the execution plan through basic writing rules. Such programs are then executed by series of MapReduce jobs.

RDFHive: Apache Hive [30] provides a mechanism to store structured data using relational tables on-top of the HDFS. For the needs of this study, we further developed RDFHive[10] to analyze how distributed relational systems behave with RDF data [1]. To this end, we load N-Triples RDF files [8] into a triple column table: one column for each RDF sentence field. Then, to evaluate SPARQL queries, we translate them into Hive-QL queries (a specific SQL-like query language) before running them. RDFHive is thereby an other member of the category of SPARQL direct evaluators.

4.3 Overall Behavior of Systems

In this Section we report on the overall behavior of each tested systems for the three datasets presented in Table 2, namely WatDiv1k, Lubm1k and Lubm10k. These datasets constitute appropriate yardsticks for studying how the tested systems behave when the dataset size grows, with the characteristics of the cluster used (*cf.* Section 4.1.3). Specifically, the WatDiv1k dataset can still be held in memory of one single VM, while the Lubm1k dataset becomes too large. Lubm10k is even larger than the whole available RAM of the cluster.

Figure 3a presents the times spent by each datastore for preprocessing the datasets[11]. Figure 3b summarizes the problematic cases. Figures 3c, 3d & 3e respectively show the elapsed times for evaluating queries over WatDiv1k, Lubm1k and Lubm10k.

We further comment on the behavior of each system pushed to the limits below, and conclude this section with comparative and more general observations.

4store: 4store achieves to load Lubm1k in around 3 hours (Figure 3a). But it spent nearly three days (69 hours) to ingest the 10 times larger dataset Lubm10k. While the progression was observed to be linear to load smaller datasets (*i.e.* a 2 times larger set was twice longer to load), 4store slowed down with a billion of triples. To execute the whole set of LUBM queries on Lubm1k (Figure 3d), 4store never spent more than one minute evaluating each query except Q1, Q2 and Q14 (respectively 64, 75 and 109 seconds). Furthermore, it achieves sub-second response time for WatDiv queries (excepting C2 and C3) with WatDiv1k (Figure 3c).

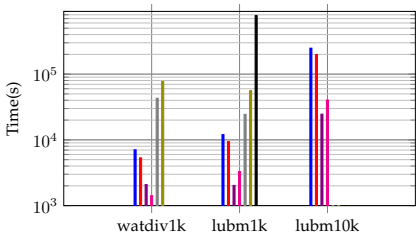
CumulusRDF: CumulusRDF is very slow to index datasets: it took almost a week only to preprocess Lubm1k (Figure 3a). By loading smaller datasets (*e.g.* Lubm100 or Lubm10), we notice that the empirical loading time is proportional to the dataset size. That is why we decided not to test it on Lubm10k which is 10 times larger. During the evaluation of the LUBM set of queries on Lubm1k (Figure 3d), the test of CumulusRDF revealed three points. (1) Q2 and Q9 which are the most difficult queries of the benchmark (see Section 4.1.1) took repectively almost 5000 seconds and 2500 seconds. (2) Q14 answered in 1600 seconds seems to slow CumulusRDF because of its large output. (3) The remaining queries were all evaluated in less than 20 seconds.

CouchBaseRDF: We recall that CouchBaseRDF is an in-memory distributed datastore, which means that datasets are distributed on the main memory of the cluster's nodes. As expected, loading Lubm10k, which is larger than the whole available RAM on the cluster, was impossible. Actually, it crashed our cluster after more than 16 days *i.e.* all the nodes were frozen; and we had to crawl the logs in order to find that it ran out of RAM and SWAP after only indexing nearly one third of the dataset. CouchBaseRDF evaluates quickly queries on Lubm1k (Figure 3d), compared to the other evaluators; but it fails answering Q2 and Q14 throwing an exception after two minutes. We also show (Figure 3c) that CouchBaseRDF is slow to evaluate C2 (about 2000 seconds) and fails with an exception evaluating C3.

RYA: RYA achieves to load WatDiv1k and Lubm1k in less than one hour and preprocesses Lubm10k in less than 10 (Figure 3a). However, we note that it needs more preprocessing time with WatDiv1k (15GB) than with Lubm1k (23GB) due to the larger number of predicates WatDiv involves. RYA was not able to answer three queries: C2 & C3 of WatDiv and Q2 of LUBM. In these cases, RYA runs indefinitely without failing or declaring a timeout. To answer the rest of the queries (Figures 3c & 3d), RYA needs less than 10 seconds for most of the LUBM queries excepting Q1, Q3 and Q14. With WatDiv1k, RYA has response times varying over three orders of magnitude *e.g.* L4 which needs 10 seconds and F3 needs 10819. Thanks to its sorted tables (on top of Accumulo), RYA is able to answer quickly queries which involving small intermediate results; therefore, it needs the same amount of time with Lubm10k (Figure 3e) than with Lubm1k (Figure 3d).

10. http://tyrex.inria.fr/rdfhive/home.html

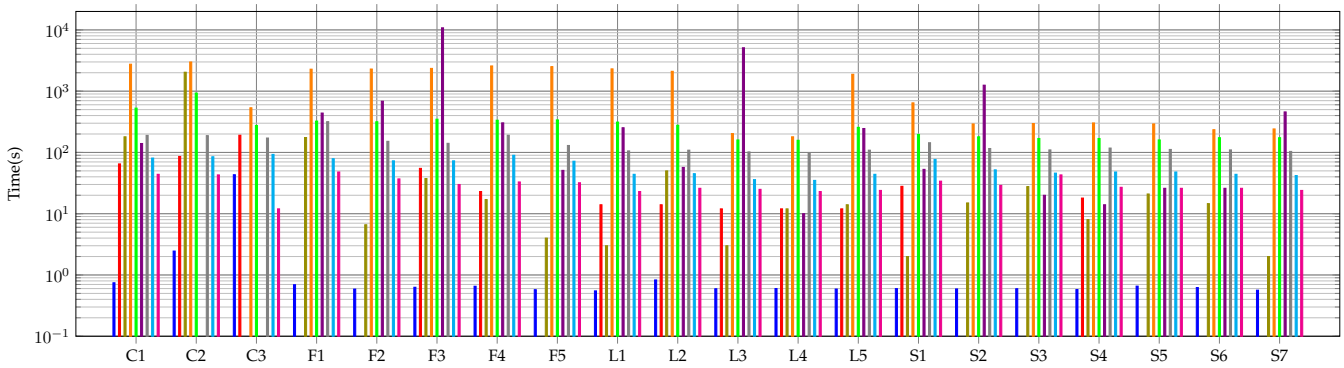
11. Times reported for the HDFS-based systems do not include the times required to import the original files on the distributed file system.



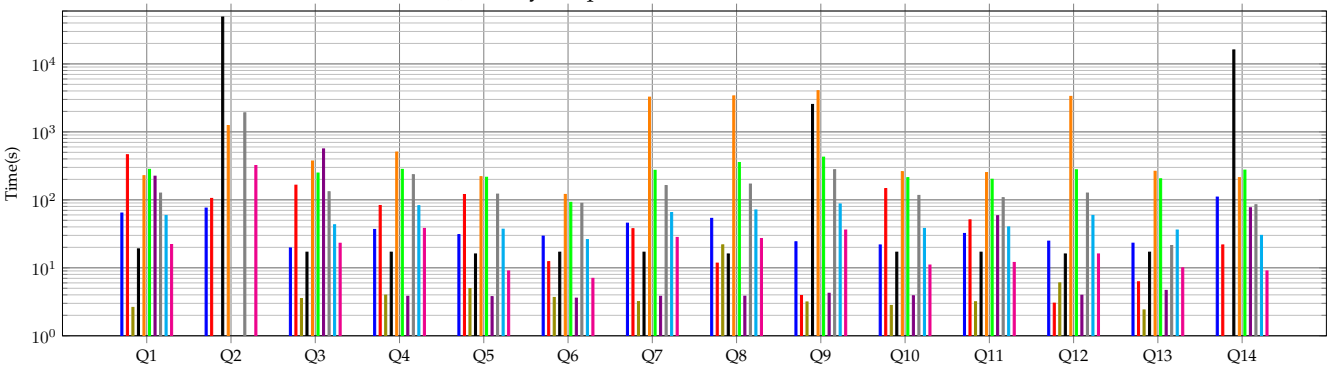
(a) Preprocessing Time.

Evaluator	WatDiv1k	Lubm1k	Lubm10k
CliqueSquare	F1,2,5 & S2,3,5,6,7 Parser	∅	∅
CouchBaseRDF	C3 Failure	Q2,14 Failure	Pre-processing Failure
RDFHive	∅	Q2 Timeout	Q2 Timeout
RYA	C2,3 Timeout	Q2 Timeout	Q2 Timeout
S2RDF	∅	∅	Pre-processing Failure
SDE	∅	Q2 Timeout	Q2 Timeout

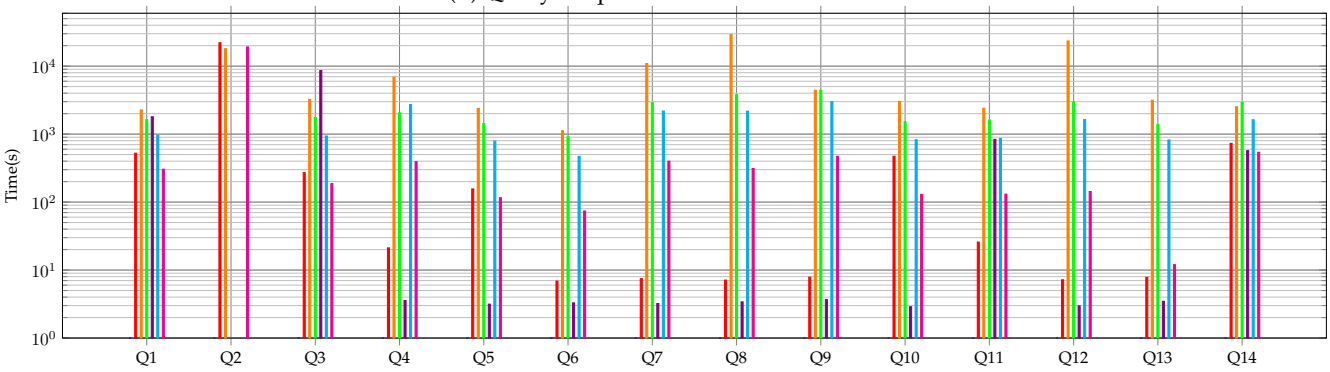
(b) Failure Summary for problematic evaluators.



(c) Query Response Time with WatDiv1k.



(d) Query Response Time with Lubm1k.



(e) Query Response Time with Lubm10k.

Fig. 3: Loading and response time with various datasets.

SPARQLGX: Thanks to its data storage model (*i.e.* the Vertical Partitioning), SPARQLGX achieved to preprocess Lubm1k in less than one hour as it does with WatDiv1k (Figure 3a). SPARQLGX preprocesses Lubm10k in about 11 hours. As shown in Figure 3d, all queries but Q2 and Q9 have been evaluated on this dataset in less than 30 seconds. Indeed, these two ones took respectively 250 and 36 seconds. Figure 3c shows that SPARQLGX always answer the WatDiv queries in less than one minute, and the average response time is 30 seconds.

S2RDF: While S2RDF was able to preprocess WatDiv1k and Lubm1k correctly (Figure 3a), it fails with Lubm10k throwing a memory space exception. Nonetheless, we also notice that preprocessing WatDiv1k was about two times longer than preprocessing Lubm1k; this counterintuitive observation can be explained by the vertical partitioning extension strategy used by S2RDF. Since it computes additional tables based on pre-computation of possible joins, it has to generate more additional table when the number of distinct predicate-object combinations increases. To evaluate WatDiv queries, S2RDF always needs less than 200 seconds excepting F1 (Figure 3c) and the average response time is 140 seconds. Figure 3d presents the S2RDF results with Lubm1k, we notice that all queries are aswered in less than 300 seconds excepting Q2 which exceeds one thousand seconds due to its large intermediate results that have to be shuffled across the cluster.

CliqueSquare: CliqueSquare achieves to load WatDiv1k, Lubm1k and Lubm10k (Figure 3a). Figures 3d & 3e show how its storage model impacts its performances compared to the other evaluators. Actually, having a large number of small files allows CliqueSquare to evaluate the LUBM queries having small intermediate results in the same temporal order of magnitude on Lubm10k as the one needed on Lubm1k (see *e.g.* Q10). We notice that CliqueSquare cannot establish a query plan for the WatDiv queries with its SPARQL parser reporting that the URIs were not "correctly formated". We finally succeeded to evaluate some queries by modifying their syntax as explained in our website. Unfortunately, it appears that we cannot hack queries having at least such a predicate: "<...#type>" (*i.e.* F1, F2, F5, S2, S3, S5, S6 and S7) unless we modify Cliquesquare's source code. Nonetheless, CliqueSquare needs 12 seconds in average to answer each WatDiv linear query, and spends more than one minute to evaluate each complex one (Figure 3c).

PigSPARQL: PigSPARQL evaluates directly the queries after a translation from SPARQL to a PigLatin sequence. Thus, there is no preprocessing phase, we just have to copy the triple file on the HDFS. As shown in Figure 3d, PigSPARQL needs more than one thousand seconds to answer queries 2, 7, 8, 9 and 12 on Lubm1k while the other queries take around 200 seconds. We observe the same behaviors when evaluating these queries on Lubm10k (Figure 3e). Similarly, the same order of magnitude applies with WatDiv1k (Figure 3c).

RDFHive: RDFHive only needs a triple file loaded on the HDFS to start evaluating queries. It appears that RDFHive was unable to answer Q2 of LUBM *i.e.* no matter the time allowed, it could not finish the evaluation. On Lubm1k (Figure 3d), we also notice that each remaining query is evaluated on Lubm1k in a 200 to 450 seconds period with a 256-second average response time. Similarly (Figure 3c), RDFHive has 289-second average response time with WatDiv1k.

SDE: Since SDE is a SPARQL direct evaluator, it does not need any preprocessing time to ingest datasets. Its average response times with WatDiv1k, Lubm1k and Lubm10k (Figures 3c, 3d & 3e) are respectively 60, 51 and 1460 seconds. We observe that the average response time with Lubm10k is about 28 times larger than the one with Lubm1k (which is 10 times larger) indeed Q4, Q7, Q8, Q9, Q12 and Q14 do not perform well because of their large intermediate results.

General Observations: A first lesson learned is that, for the same query on the same dataset, elapsed times can differ very significantly (the time scale being logarithmic) from one system to another (as shown for instance on Figure 3d).

Interestingly, we also observe that, even with large datasets, most queries are not harmful *per se*, *i.e.* queries that incurr long running times with some implementations still remain in the "comfort zone" for other implementations, and sometimes even representing a case of demonstration of efficiency for others. For example, the response times for Q12 with Lubm1k (see Figure 3d) span more than 3 orders of magnitude. Interestingly and more generally, for each query, there is at least a difference of one order of magnitude between the times spent by the fastest and the slowest evaluators.

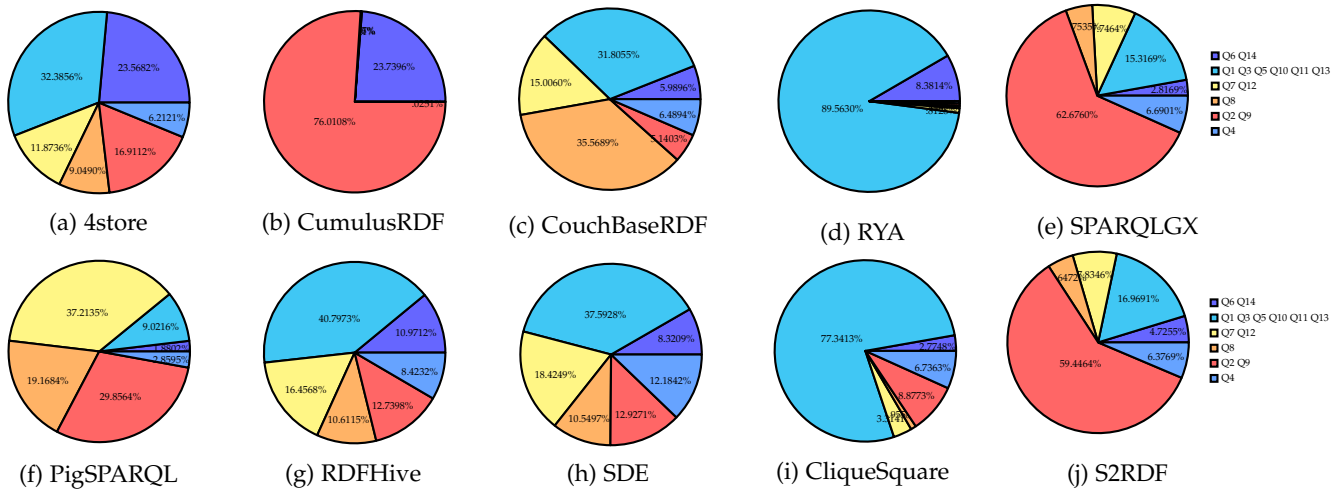
These observations gave rise to the further comparative analysis guided by criteria (and supplemented with additional metrics) that we present in Section 4.4.

4.4 Comparative Analysis Driven By Features

The variety of RDF application workloads makes it hard to capture how well a particular system is suited compared to the others in a way based exclusively on time measurements. For instance, consider these five features that have different needs and where the main emerging requirement is not the same:

- *Velocity*: applications might favour the fastest possible answers (even if that means storing the whole dataset in RAM, when possible).
- *Immediacy*: applications might need to evaluate some SPARQL queries only once. This is typically the case of some pipeline extraction applications that have to extract data cleaned only once.
- *Dynamicity*: applications might need to deal with dynamic data, requiring to react to frequent data updates. In this case a small preprocessing time (or the capacity to react to updates in an incremental manner) is important.
- *Resiliency*: applications that process very large data sets (spanning accross many machines) with complex queries (taking *e.g.* days to complete) might favour forms of resiliency for trying to avoid as much as possible to recompute everything when a machine fails because it is likely to happen.

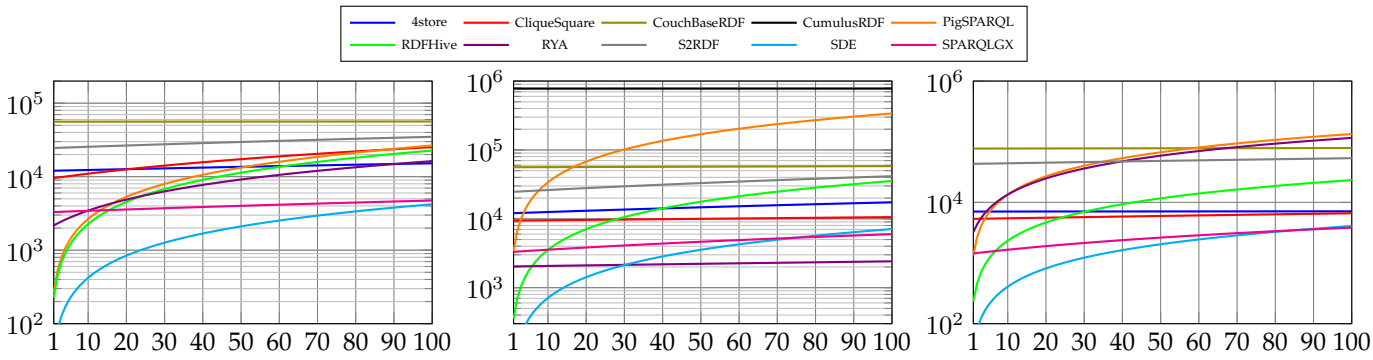
Since many applications actually combine these requirements by affecting more or less importance to each, we believe that they represent a good basis on which to compare



(a) 4store (b) CumulusRDF (c) CouchBaseRDF (d) RYA (e) SPARQLGX

(f) PigSPARQL (g) RDFHive (h) SDE (i) CliqueSquare (j) S2RDF

Fig. 4: Time distributions with Lubm1k.



(a) lubm1k: Q1,Q3,Q5,Q10,Q11,Q13 (b) lubm1k: Q8 (c) watdiv1k: L1,L2,L3,L4,L5

Fig. 5: Tradeoff between preprocessing and query evaluation times (seconds).

the tested systems. In this Section, we thus further compare the tested stores by analysing the metrics introduced in Section 4.1.2 according to the five aforementioned requirements. For the sake of brevity, we will directly refer to these requirements as "velocity", "immediacy", "dynamicity" and "resiliency" in the rest of the paper.

4.4.1 *Velocity* The Faster, The Better

Figure 3d shows the time per query using Lubm1k as dataset for each tested store. The logarithmic scale allows to easily observe the various magnitude orders required to execute queries. It is then possible to notice significant differences between *e.g.* CumulusRDF that needs more than 10^4 seconds to answer Q2 or Q14 while for instance 4store always has response times included in $[10, 100]$ seconds. More generally, it appears that Q2 incurrs the longest response times because of its triangular pattern and its large intermediate results. If we compute the sum of the response times for all the queries of Lubm1k for each evaluator, we notice that our candidates have performances spanning over three orders of magnitude from 568 seconds with SPARQLGX and 67718 seconds with CumulusRDF. Thereby, to execute the whole set of 14 LUBM queries, SPARQLGX and 4store constitute the fastest solutions.

In addition, Figure 3d also shows that some stores seem to behave similarly (according to the time metric alone) with some queries *e.g.* PigSPARQL needs the same order of magnitude for evaluating Q2, Q7, Q8, Q9, Q12. That is why we group LUBM queries by their graph variables (introduced in Table 3) in Figure 4 to represent time distributions for each store, excluding failed queries listed in Figure 3b. For instance, Figure 4b shows that 99% of the CumulusRDF time is consumed by the evaluation of Q2, Q9 and Q14. This representation also allows to notice similarities between stores, for example we show that because they both rely on Apache Spark, S2RDF (Figure 4j) and SPARQLGX (Figure 4e) present the same distributions; indeed, their joining method is common even if S2RDF uses the SparkSQL layer. The time distributions also highlight that RDFHive (Figure 4g) and SDE (Figure 4h) which are both direct evaluators share similar pies: because (1) they have to read the entire source file before joining and (2) their join plans are the same since they do not order triple patterns prior to the execution. Figure 4f shows that PigSPARQL is essentially slow for evaluating Q2,Q7,Q8,Q9,Q12 (≈85% of the time); in fact, we discover that PigSPARQL is slow if there are stricly more than two joins involved in the query.

More generally, this discussion around the variable graphs highlights the RDF storage methods implemented

by the considered SPARQL evaluators presented in Table 5 and classified in literature in *e.g.* [20]. SPARQLGX and S2RDF both share similar pie-charts and vertical partitioning on top of Apache Spark. The *triple table* approach adopted by RDFHive and SDE also provides similar charts since these evaluators have to read at least once the whole dataset before starting to join.

4.4.2 *Immediacy* Preprocessing is Investing

The preprocessing time required before querying can be seen as an investment *i.e.* taking time to preprocess data (load/index) should imply faster query response time, offseting the time spent in preprocessing. To illustrate when the trade-off is really worth, Figure 5 presents the preprocessing costs for Lubm1k and WatDiv1k in various cases related to the query types presented in Table 3. In other words, we draw on a logarithmic time scale for each evaluator the affine line $y = ax + b$ where a is the average time required to evaluate one of the considered queries and where b is the preprocessing time; for instance in Figure 5c, a will represent the average time to evaluate one WatDiv linear query.

Among competitors, we distinguish the set of "direct evaluators" (See Table 5) that are capable of evaluating SPARQL queries at no preprocessing cost (they do not require any preprocessing of RDF data): PigSPARQL, RDFHive and SDE. As shown in Figure 5, SDE outperforms all the other datastores if less than 20 queries are evaluated. Beyond this threshold, SPARQLGX or RYA become more interesting. In addition, we also notice that in some cases (for instance Q8, see Figure 5b) PigSPARQL provide worse performances than RYA or SPARQLGX all the time.

These statements are also related to RDF storage approaches; indeed, the more complex it is, the less immediacy-efficient the evaluator is. As a consequence, we can rank for this feature the various storage methods from the best ones: first the schema-carfree triple table of the direct evaluators, next the vertical partitioning, then the key-value table (*e.g.* RYA) and finally the complicated indexing methods.

4.4.3 *Dynamicity* Changing Data

We now examine how the tested stores can be set up to react to frequent data changes. The W3C proposes an extension of SPARQL to deal with updates[12]. Instead of re-loading all the datasets after each single change, some solutions can be set up to load bulks of updates. To the best of our knowledge, there is no widely-used benchmark dealing exclusively with the SPARQL Update extension. That is why we develop a basic experimental protocol based on both LUBM and WatDiv benchmarks. It can be divided into three steps: **(1)** We load a large dataset *i.e.* Lubm1k (Table 2) and evaluate the simple LUBM query Q1 then we measure performances for preprocessing and query evaluation. **(2)** We add a few RDF triples to modify the output of Q1; we run again Q1 and then remove the freshly added triples while measuring the time for each step. **(3)** Finally, we reproduce the previous step with a larger number of triples using WatDiv1 (which contains about one hundred thousand triples) and querying with C1. Although simple, our protocol allows testing the several features such as inserting/deleting a few triples and a large bulk of triples. The benchmarked datastores exhibit various behaviors. First, the direct evaluators (*e.g.* PigSPARQL, RDFHive and SDE) evaluate queries without requiring a preprocessing phase. In that case, updating a dataset boils down to editing a file on the HDFS and retriggering query evaluation. Second, other datastores simply do not implement any support (even partial) of updates. This category of stores (*e.g.* S2RDF, CumulusRDF, CouchBaseRDF, RYA or CliqueSquare) thus forces the reprocessing of the whole dataset. Third, some of the benchmarked datastores are able to deal with dynamic datasets *i.e.* 4store and SPARQLGX. 4store implements the SPARQL Update extension whereas SPARQLGX offers a set of primitives to add or delete sets of triples. Moreover, unlike 4store, SPARQLGX is also able to delete in one action a large set of triples, whereas 4store needs to execute several "Delete Data"-processes if the considered set cannot fit in memory.

12. https://www.w3.org/Submission/SPARQL-Update/

Systems	Lubm1k (GB)	WatDiv1k (GB)
S2RDF	**13.057**	15.150
RYA	16.275	**11.027**
CumulusRDF	20.325	–
4store	20.551	14.390
CouchBaseRDF	37.941	20.559
SPARQLGX	39.057	23.629
CliqueSquare	55.753	90.608
PigSPARQL	72.044	46.797
RDFHive	72.044	46.797
SDE	72.044	46.797

TABLE 6: Disk Footprints (including replication).

4.4.4 *Resiliency* Having Duplicates

Data Resiliency: When an application processes a very large dataset stored across many machines, it is interesting for the system to implement some level of tolerance in case a datanode is lost. To implement data resilience, stores typically replicate data across the cluster which implies a larger disk footprint. For our experiments, we stick to the default replication parameters. As a consequence, the HDFS-based systems have their data replicated twice and provide some level of data resilience. Table 6 presents the effective disk footprints (including replication) with Lubm1k and WatDiv1k where the HDFS-based systems are outlined in gray. Due to their preprocessing methods, we note that S2RDF and CliqueSquare need more disk space to store WatDiv1k than Lubm1k whereas this last one is larger (see Table 2). Furthermore, counterintuitively, it appears that evaluators having replicated data can have lighter disk footprints than not-replicated ones *e.g* S2RDF and RYA versus CouchBaseRDF.

Computation Resiliency: If an application has to evaluate complex queries (taking *e.g.* days), it is interesting for the system not to be forced to compute everything from scratch whenever a machine becomes unreachable. This situation is liketly to happen for a variety of reasons (*e.g.* reboot, failure, network latency). The tested systems exhibit several behaviours when a machine fails during computation. For stores having no data replication, the loss of any machine can stop the computation if the lost data fragment is mandatory; thus some stores fail when a machine is lost: 4store and CumulusRDF; whereas CouchBaseRDF adopts

another method waiting seven minutes until the return of the machine. More generally, the HDFS-based triplestores cannot lose mandatory fragments of data, thereby RDFHive, SPARQLGX, SDE, RYA, and CliqueSquare still succeed when one (or even two) machine fails during computation; however, PigSPARQL waits indefinitely the return of the lost partition. For stores having a master/slave structure *e.g.* SPARQLGX, the loss of the node hosting the master process prevents any result to be obtained. From our tests, only two different methods successfully faced a loss of worker nodes: (1) waiting for their returns *e.g.* CouchBaseRDF and PigSPARQL; (2) using the remaining nodes and benefiting from data replication *e.g.* CliqueSquare, RDFHive, RYA, S2RDF, SDE, SPARQLGX.

5 Related Work

In recent years, many RDF systems capable of evaluating SPARQL queries have been developed [19]. These stores can be divided in two categories: centralized systems (*e.g.* RDF-3X [31] or Virtuoso [32]) and distributed ones, that we further review. Distributed RDF stores can in turn be divided into three categories. (1) The *ad-hoc* systems that are specially designed for RDF data and that distribute and store data across the nodes according to custom ad-hoc methods (*e.g.* 4store [21]). (2) Other systems use a communication layer between centralized systems deployed across the cluster and then evaluate sub-queries on each node such as Partout with RDF-3X [33]. (3) Lastly, some RDF systems [27], [24], [28], [25] are built on top of distributed Cloud platforms such as Apache Hadoop. One major interest of such platforms relies on their common file systems (*e.g.*, HDFS): indeed various applications can access data at the same time and the distribution/replication issues are transparent. These systems [27], [24], [28], [25], then evaluate SPARQL conjunctive queries using various tools as presented in Section 4 (*e.g.* Accumulo, Hive, Spark, etc.). To set up appropriate tools for pipeline applications, we choose to distribute data with a Cloud platform (HDFS) and evaluate queries using Spark. We compared the performances of SPARQLGX with the most closely related implementations in Section 4.

Finally, it is worthwhile to notice that SPARQL is a very expressive language which offers a rich set of features and operators. Most evaluators based on Cloud platforms focus on the restricted SPARQL fragment composed of conjunctive queries. SPARQLGX also natively supports a slight extension of this fragment with UNION and OPTIONAL operators at top level.

The experimental validation part of this work benefited from the extensive earlier works on benchmarks for RDF systems. There are many benchmarks designed for evaluating RDF systems [34], [35], [36], [14], [37], [38], [13], [17]. Some of them are particularly popular: LUBM [14], WatDiv [13], SP^{2}Bench [38], DBpedia Bench [36], BSBM [37], and RBench [17]. We notably reused LUBM [14] and WatDiv [13] for testing the BGP fragment, and because we wanted deterministic data generators for ensuring reproducibility of our results. Compared to all these works, we focus on testing distribution techniques by considering a set of 10 state-of-the-art implementations; see *e.g.* [19], [20] for recent surveys about distributed RDF datastores and their storage approaches. Compared to studies included in the aforementioned benchmarks, we consider more competing implementations on a common ground. Furthermore, while earlier works on RDF benchmarks exclusively focused on measuring elapsed times (and sometimes disk footprints), we measure a broader set of indicators encompassing *e.g.* network usage. This allows to refine the comparative analysis according to features and requirements from a slightly higher perspective, as discussed in Section 4.4. This also allows to more precisely identify the bottlenecks of each system when they are pushed to the limits. The experimental part of this work was also inspired by the empirical study carried out by Cudré *et al.* where five distributed RDF datastores using various NoSQL backends were evaluated [18]. It is worthwhile to notice that our work does not invalidate earlier results but supplement them with more results. In particular, in the present work, we update the list of evaluators (we consider more of them, with more recent ones, not limited to the simple addition of SPARQLGX) and we also focus on ranking the candidates depending on various features thanks to the broader set of metrics we analysed.

6 Conclusion

We propose SPARQLGX: a tool for the efficient evaluation of SPARQL queries on distributed RDF datasets. SPARQL queries are translated into Spark executable code, which is optimized to leverage the advantages of the Spark platform in the specific setting of RDF data. SPARQLGX also comes with a direct evaluator based on the same SPARQL translation process and called SDE, for situations where preprocessing time matters at least as much as query evaluation time.

We reported on an experimental evaluation of SPARQLGX in comparison with 9 state-of-the-art distributed SPARQL evaluators. By analysing a broad set of metrics, we pushed the comparison further than traditional experimental evaluations that focus only on running times. We considered five dimensions of comparison that help in clarifying the limitations and advantages of each SPARQL evaluator according to the different requirements met in practical use cases. Experimental results indicate that SPARQLGX outperforms several state-of-the-art Hadoop-reliant systems, while implementing a simple architecture that is easily deployable across a cluster.

References

[1] D. Graux, L. Jachiet, P. Genevès, and N. Layaïda, "SPARQLGX: Efficient Distributed Evaluation of SPARQL with Apache Spark," *To appear in ISWC*, 2016.

[2] P. Hayes and B. McBride, "RDF semantics," *W3C Rec.*, 2004.

[3] "SPARQL 1.1 overview," March 2013, http://www.w3.org/TR/sparql11-overview/.

[4] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," *NSDI*, 2012.

[5] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[6] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–10.

[7] M. Odersky, "The scala language specification v 2.9," 2014.

[8] "RDF 1.1 N-Triples: A line-based syntax for an RDF graph," 2014, http://www.w3.org/TR/n-triples/.

[9] Abadi, Marcus, Madden, and Hollenbach, "Scalable semantic web data management using vertical partitioning," *VLDB*, 2007.

[10] M. A. Gallego, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente, "An empirical study of real-world SPARQL queries," in *1st International Workshop on Usage Analysis and the Web of Data at the 20th International World Wide Web Conference*, 2011.

[11] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, "DBpedia: A nucleus for a web of open data," *The semantic web*, pp. 722–735, 2007.

[12] J. D. Fernández, C. Gutierrez, and M. A. Martínez-Prieto, "RDF compression: basic approaches," in *Proceedings of the 19th international conference on World wide web*. ACM, 2010, pp. 1091–1092.

[13] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee, "Diversified stress testing of RDF data management systems," in *ISWC*. Springer, 2014, pp. 197–212.

[14] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems," *Web Semantics*, 2005.

[15] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer, "DBpedia - A large-scale, multilingual knowledge base extracted from wikipedia," *Semantic Web*, vol. 6, no. 2, pp. 167–195, 2015. [Online]. Available: https://doi.org/10.3233/SW-140134

[16] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and complexity of SPARQL," *ACM Trans. Database Syst.*, vol. 34, no. 3, pp. 16:1–16:45, Sep. 2009. [Online]. Available: http://doi.acm.org/10.1145/1567274.1567278

[17] S. Qiao and Z. M. Özsoyoğlu, "Rbench: Application-specific RDF benchmarking," in *SIGMOD*. ACM, 2015, pp. 1825–1838.

[18] P. Cudré-Mauroux, I. Enchev, S. Fundatureanu, P. Groth, A. Haque, A. Harth, F. L. Keppmann, D. Miranker, J. F. Sequeda, and M. Wylot, "NoSQL databases for RDF: An empirical evaluation," *ISWC*, pp. 310–325, 2013.

[19] Z. Kaoudi and I. Manolescu, "RDF in the clouds: a survey," *The VLDB Journal*, vol. 24, no. 1, pp. 67–91, 2015.

[20] D. C. Faye, O. Curé, and G. Blin, "A survey of RDF storage approaches," *Arima Journal*, vol. 15, pp. 11–35, 2012.

[21] S. Harris, N. Lamb, and N. Shadbolt, "4store: The design and implementation of a clustered RDF store," *SSWS*, 2009.

[22] G. Ladwig and A. Harth, "CumulusRDF: linked data management on nested key-value stores," *SSWS 2011*, p. 30, 2011.

[23] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[24] R. Punnoose, A. Crainiceanu, and D. Rapp, "RYA: a scalable RDF triple store for the clouds," in *International Workshop on Cloud Intelligence*. ACM, 2012, p. 4.

[25] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen, "S2RDF: RDF querying with SPARQL on spark," *VLDB*, pp. 804–815, 2016.

[26] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark SQL: Relational data processing in spark," in *SIGMOD*. ACM, 2015, pp. 1383–1394.

[27] F. Goasdoué, Z. Kaoudi, I. Manolescu, J.-A. Quiané-Ruiz, and S. Zampetakis, "Cliquesquare: Flat plans for massively parallel RDF queries," in *ICDE*. IEEE, 2015, pp. 771–782.

[28] A. Schätzle, M. Przyjaciel-Zablocki, and G. Lausen, "PigSPARQL: Mapping SPARQL to pig latin," in *Proceedings of the International Workshop on Semantic Web Information Management*. ACM, 2011, p. 4.

[29] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *SIGMOD*. ACM, 2008, pp. 1099–1110.

[30] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[31] T. Neumann and G. Weikum, "RDF-3X: a RISC-style engine for RDF," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 647–659, 2008.

[32] O. Erling and I. Mikhailov, *Virtuoso: RDF support in a native RDBMS*. Springer, 2010.

[33] L. Galarraga, K. Hose, and R. Schenkel, "Partout: A distributed engine for efficient RDF processing," in *Proceedings of the companion publication of the 23rd international conference on World wide web companion*. International World Wide Web Conferences Steering Committee, 2014, pp. 267–268.

[34] R. Angles, P. Boncz, J. Larriba-Pey, I. Fundulaki, T. Neumann, O. Erling, P. Neubauer, N. Martinez-Bazan, V. Kotsev, and I. Toma, "The linked data benchmark council: a graph and RDF industry benchmarking effort," *ACM SIGMOD Record*, vol. 43, no. 1, pp. 27–31, 2014.

[35] G. Demartini, I. Enchev, M. Wylot, J. Gapany, and P. Cudré-Mauroux, "Bowlognabench – Benchmarking RDF Analytics," in *International Symposium on Data-Driven Process Discovery and Analysis*. Springer, 2011, pp. 82–102.

[36] M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo, "DBpedia SPARQL Benchmark – Performance assessment with real queries on real data," *ISWC*, pp. 454–469, 2011.

[37] C. Bizer and A. Schultz, "The berlin SPARQL benchmark," *IJSWIS*, 2009.

[38] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel, "SP^{2}Bench: a SPARQL performance benchmark," *ICDE*, pp. 222–233, 2009.